CS 536

Optimization
• Last time:
  – CodeGen for the remainder of AST nodes
  – Introduced the control-flow graph

• This time:
  – Optimization Overview
  – Discuss a couple of optimizations
    • Review CFGs
  – Course evaluations
Optimization Overview
Optimization Goals

• What are we trying to accomplish?
  – Traditionally, speed
  – Lower power
  – Smaller footprint
  – Bug resilience?

• The fewer instructions the better
• Informally: Don’t change the program’s output
  – We may relax this to “Don’t change the program’s output on good input”
  – This can actually be really hard to do
• There’s no perfect way to check equivalence of two arbitrary programs
  – If there was we could use it to solve the halting problem
  – We’ll attempt to perform behavior-preserving transformations
• A perspective on optimization
  – Recognize some behavior in a program
  – Replace it with a “better” version

• Constantly plagued by the halting problem
  – We can only use approximate algorithms to recognize behavior
• Two terms in program analysis / behavior detection:
  – Soundness: All results that are output are valid
  – Completeness: All results that are valid are output
• These terms are necessarily mutually exclusive
  – If an algorithm was sound and complete, it would either:
    1. Solve the halting program
    2. Detect a trivial property
• We want our optimizations to be *sound* transformations
  – In other words, they are always valid, but will miss some behaviors
You may be thinking...

• I’m sad because this makes optimization seem pretty limited

• Cheer up! Our optimization may be able to detect many *practical* instances of the behavior
Now you may be thinking...

• I’m happy because I’m guaranteed that my optimization won’t do any harm.

• Settle down! Our optimization still needs to be efficient.
Or maybe you are thinking...

• I don’t know how to feel about any of this without understanding how often it comes up
What *Can* We Do?

• We can pick some low-hanging fruit
Example
Optimizations
• A naïve code generator tends to output some silly code
  – Err on the side of correctness over efficiency
• Pattern-match the most obvious problems
• Consider the following sequence of instructions:

```
push  
  sw  $t0  0($sp)
  subu $sp $sp 4
pop   
  lw  $t0  4($sp)
  addu $sp $sp 4
```

• We’d like to remove this sequence...
  – Is it sound to do so?
  – Maybe not!
• Program as a flowchart
• Nodes are “Basic Blocks”
• Edges are control transfers
  – Fallthrough
  – Jump
  – *Maybe* function calls
• We can limit our peephole optimizations to *intra-block* analysis
  – This ensures, by definition, that no jumps will intrude on the sequence

• We will assume for the rest of our peephole optimizations that instruction sequences are in one block
• Called “peephole” optimization because we are conceptually sliding a small window over the code, looking for small patterns
• Four different optimizations
  – Peephole optimization
  – Loop-Invariant Code Motion
  – For-loop strength reduction
  – Copy propagation

Performed after machine code generation

Performed before machine code generation
• Remove no-op sequences
  – Push followed by pop
  – Add/sub 0
  – Mul/div 1

push

pop

addu $t1 $t1 0

mul $t2 $t2 1
• Simplify sequences
  – Ex. Store then load
  – Strength reduction

```
sw $t0 -8($fp)
lw $t0 -8($fp)
mul $t1 $t1 2
add $t2 $t2 1
```

Useless instruction
shift-left $t1
inc $t2
• Jump to next instruction
• Loop Invariant Code Motion
  – Don’t duplicate effort in a loop

• Goal
  – Pull code out of the loop
  – “Loop hoisting”

• Important due to “hot spots”
  – Most execution time due to small regions of deeply-nested loops
for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        for (k=0; k<100; k++) {
            \[ A[i][j][k] = i*j*k \]
        }
    }
}

Sub-expression invariant with respect to Innermost loop

for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        temp = i * j
        for (k=0; k<100; k++) {
            A[i][j][k] = temp *k
        }
    }
}
LICM: When Should we Do it?

- In the previous example, showed LICM on source code
- At IR level, more candidate operations
- ASM might be *too* low-level
  - Need a guarantee that the loop is *natural*
  - No jumps into the loop

```c
tmp0 = FP + offsetA
for (i=0; i<100; i++){
    tmp1 = tmp0 - i*40000
    for (j=0; j<100; j++){
        tmp2 = ind2
        tmp3 = i*j
        for (k=0; k<100; k++){
            T0 = tmp3 * k
            T1 = tmp2 - k*4
            store T0, 0(T1)
        }
    }
}
```
LICM: How Should we Do it?

• Two factors, which really generalize to optimization:
  – Safety
    • Is the transformation semantics-preserving?
      – Make sure the operation is truly loop-invariant
      – Make sure ordering of events is preserved
  – Profitability
    • Is there any advantage to moving the instruction?
      – May end up doing instructions that are never executed
      – May end up performing more intermediate computation than necessary
Other Loop Optimizations

• Loop unrolling
  – For a loop with a small, constant number of iterations, we may actually save time by just placing every copy of the loop body in sequence (no jumps)
  – May also consider doing multiple iterations within the body

• Loop fusion
  – Merge two sequential, independent loops into a single loop body (fewer jumps)
Jump Optimizations

Disclaimer: Require some extra conditions

• Jump around jump

\[
\text{beq } \$t0, \$t1, \text{Lab1} \\
\text{j Lab2} \\
\text{Lab1: } \ldots \\
\ldots \\
\text{Lab2: } \ldots
\]

\[
\text{bne } \$t0, \$t1, \text{Lab2} \\
\text{Lab1: } \ldots \\
\ldots \\
\text{Lab2: } \ldots
\]

• Jump to jump

\[
\text{j Lab1} \\
\ldots \\
\text{Lab1: } \text{j Lab2} \\
\ldots \\
\text{Lab2: } \ldots
\]

\[
\text{j Lab2} \\
\ldots \\
\text{Lab1: } \text{j Lab2} \\
\ldots \\
\text{Lab2: } \ldots
\]
• The past two optimizations had some caveats
  – There may be a jump into your eliminated code

• We’d like to introduce a control-flow concept beyond basic blocks:
  – Guarantee that block1 must be executed in order to get to block2
    • This goes by a pretty boring name
• We say that block A dominates block B if A \textbf{must} be executed before B is executed.

• We say that block A postdominates block B if A \textbf{must} be executed after B.
Do we really need semantics preserving optimizations?

Are there examples where we don’t?
Next Time

- Wrap up optimization